Conditional expressions

Sometimes, *the value an expression should take depends on some condition.*

E.g. expressions should take one value under some conditions, and other values under other conditions.

Example: taking the absolute value of $x$.

$$|x| = \begin{cases} 
-x & \text{when } x < 0 \\
  x & \text{when } x \geq 0 
\end{cases}$$
In Racket, we can compute $|x|$ with the expression

$$\text{(cond } [(< x 0) (− x)]
\quad [(≥ x 0) \ x])$$

- Conditional expressions use the special form `cond`.
- *Each argument is a question/answer pair.*
  - The question is a Boolean expression.
  - The answer is a possible value of the conditional expression.
- Square brackets used by convention, for readability.
- Square brackets and parentheses are equivalent in the teaching languages (but they must be nested properly).
- Note: `abs` is a built-in function in Racket.
The *general form of a conditional expression* is ...

```lisp
(cond [question1 answer1]
      [question2 answer2]
      . . .
      [questionk answerk]) ;; where questionk could be else
```

- The questions are *evaluated in top-to-bottom order*.
- As soon as *one question is found that evaluates to true*, *no further questions are evaluated*.
- Only one answer is ever evaluated.  
  \[ \Rightarrow \text{Either the one paired with the first question that evaluates to true or the one paired with else (if it is present and reached).} \]
Example

\[ f(x) = \begin{cases} 
0 & \text{when } x = 0 \\
x \sin\left(\frac{1}{x}\right) & \text{when } x \neq 0
\end{cases} \]

(define (f x)
  (cond [(= x 0) 0]
        [else (* x (sin (/ 1 x)))]))
Simplifying conditional functions

Sometimes a question can be simplified by knowing that *if it is asked, all previous questions have evaluated to false.*

Here are the common recommendations on which course to take after CS 135, based on the mark earned.

- 0% ≤ mark < 40%: CS 115 is recommended
- 40% ≤ mark < 50%: CS 135 is recommended
- 50% ≤ mark < 60%: CS 116 is recommended
- 60% ≤ mark: CS 136 is recommended
cond without simplification

We might write the tests for the four intervals this way:

(define (course-after-cs135 grade)
  (cond [(< grade 40) 'cs115]
        [(and (>= grade 40) (< grade 50)) 'cs135]
        [(and (>= grade 50) (< grade 60)) 'cs116]
        [(>= grade 60) 'cs136]))

This method *does not take into account* that if the computation gets to the second condition we know that the grade is greater than or equal to 40% ⇒ so *it can be simplified*. 
cond with simplification

We can simplify three of the tests.

(define (course-after-cs135 grade)
  (cond [(< grade 40) 'cs115]
    [(< grade 50) 'cs135]
    [(< grade 60) 'cs116]
    [else 'cs136]))

These simplifications become second nature with practice.
Tests for conditional expressions

• Write at least *one test for each possible question/answer pair* in the expression.

• That test should be simple and direct, aimed at testing that answer.

• When the problem contains *boundary conditions* (like the cut-off between passing and failing), they *should be tested explicitly*.

• DrRacket highlights unused code.

• Properly tested code should have no highlights (i.e. no unused/untested code).
Tests for conditional expressions

For the example above:

\[
\text{(define (course-after-cs135 grade)}
  \text{(cond \[(< \text{grade 40} ) \ 'cs115]
            \[(< \text{grade 50} ) \ 'cs135]
            \[(< \text{grade 60} ) \ 'cs116]
            \[\text{else } \ 'cs136\]
  \))}
\]

there are four intervals and three boundary points, so seven tests are required (for instance, 30, 40, 45 50, 55, 60, 70).
Tests for Boolean Expressions

Testing and and or expressions is similar.

For \((\text{and} \ (\text{not} \ (\text{zero?} \ x)) \ (\leq \ (/ \ y \ x) \ c))\), we need three test cases:

1. one test case where \(x\) is zero
   (first argument to and is false)

2. one test case where \(x\) is nonzero and \(y/x > c\),
   (first argument is true but second argument is false)

3. one test case where \(x\) is nonzero and \(y/x \leq c\).
   (both arguments are true)
Types of Tests

Some of your tests, including your examples, will have been defined before the body of the function was written.

These are known as black-box tests, because they are not based on details of the code.

Other tests may depend on knowledge of the code, for example, to check specific answers in conditional expressions.

These are known as white-box tests. Both types of tests are important.
Writing tests

The textbook writes tests in this fashion:

\[(\text{= (sum-of-squares 3 4) 25})\]

which works outside the teaching languages.

\textbf{check-expect} was added to the teaching languages after the textbook was written. \textit{You should use it for all tests.}
Example: computing taxes

Purpose: Compute the Canadian tax payable on a specified income.

Examples:

Google “Canada income tax” For 2017:

• 15% on the amount in [$0 to $45,916]
• 20.5% on the amount in ($45,916 to $91,831]
• 26% on the amount in ($91,831 to $142,353]
• 29% on the amount in ($142,353 to $202,800]
• 33% on the amount over $202,800
The “piecewise linear” nature of the graph complicates the computation of tax payable.

One way to do it uses the **breakpoints** ($x$-value or salary when the rate changes) and **base amounts** ($y$-value or tax payable at breakpoints).

This is what the paper Canadian tax form does.
**Examples: Calculating the tax due**

- 15% on the amount in \([0 to 45,916]\]
- 20.5% on the amount in \((45,916 to 91,831]\]
- ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>0.15 \times 45,000 = 6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>0.15 \times 45,916 + 0.205 \times (50,000 - 45,916) = 7,724.62</td>
</tr>
</tbody>
</table>

Note the notation \([0 to x]\) means including \(x\) and the notation \((x to y]\) means not including \(x\).
Example: Calculating the tax due

- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- 26% on the amount in ($91,831 to $142,353]
- ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100,000</td>
<td>$18,423.915</td>
</tr>
</tbody>
</table>

$100,000 \times 0.15 \times $45,916 +
0.205 \times ($91,831 - $45,916) +
0.26 \times ($100,000 - $91,831) = $18,423.915

Replace $45,916 and $91,831 with constants bp1 and bp2.
Replace 15%, 20.5% and 26% with constants rate1, rate2 and rate3.
Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>( \text{rate}_1 \times $45,000 = $6,750 )</td>
</tr>
<tr>
<td>$50,000</td>
<td>( \text{rate}_1 \times \text{bp}_1 + \text{rate}_2 \times ($50,000 - \text{bp}_1) = $7,724.62 )</td>
</tr>
<tr>
<td>$100,000</td>
<td>( \text{rate}_1 \times \text{bp}_1 + \text{rate}_2 \times (\text{bp}_2 - \text{bp}_1) + \text{rate}_3 \times ($100,000 - \text{bp}_2) = $18,423.915 )</td>
</tr>
</tbody>
</table>

Now let base\(_1\) = \( \text{rate}_1 \times \text{bp}_1 \)

and let base\(_2\) = \( \text{rate}_1 \times \text{bp}_1 + \text{rate}_2 \times (\text{bp}_2 - \text{bp}_1) \), etc. for base\(_3\), ...
Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>rate1 × $45,000 = $6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>base1 + rate2 × ($50,000 - bp1) = $7,724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>base2 + rate3 × ($100,000 - bp2) = $18,423.915</td>
</tr>
</tbody>
</table>

With this plan in mind, we can begin to create the function that calculates the Canadian income tax for any income (i.e. for all five different rates).
Examples:

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$0.15 \times 45000 = 6750</td>
</tr>
<tr>
<td>$50,000</td>
<td>$0.15 \times 45916 + 0.205 \times (50000-45916) = 7724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>$0.15 \times 45916 + 0.205 \times (91831-45916) + 0.26 \times (100000-91831) = 18423.915</td>
</tr>
</tbody>
</table>

(check-expect (tax-payable 45000) 6750)
(check-expect (tax-payable 50000) 7724.62)
(check-expect (tax-payable 100000) 18423.915)
Definition header & contract

;; tax-payable: Num → Num
;; requires: income ≥ 0

(define (tax-payable income) . . . ...)

Finalize purpose

;; (tax-payable income) computes the 2017 Canadian tax payable
;; on income.
Write function body

Some constants will be useful. Put these before the purpose and other design recipe elements.

;; Rates
(define rate1 0.15)
(define rate2 0.205)
(define rate3 0.26)
(define rate4 0.29)
(define rate5 0.33)

;; Breakpoints
(define bp1 45916)
(define bp2 91831)
(define bp3 142353)
(define bp4 202800)
Instead of putting the base amounts into the program as numbers (as the tax form does), we can compute them from the breakpoints and rates.

;; basei is the base amount for interval [bpi,bp(i+1)]
;; that is, tax payable at income bpi

(define base1 ( * ( - bp1 0) rate1 ))
(define base2 ( + base1 ( * ( - bp2 bp1) rate2)))
(define base3 ( + base2 ( * ( - bp3 bp2) rate3)))
(define base4 ( + base3 ( * ( - bp4 bp3) rate4))))
;; tax-payable: Num → Num
;; requires: income ≥ 0
(define (tax-payable income)
  (cond [(< income 0) 0] ;; Not strictly necessary given contract
        [(< income bp1) (∗ income rate1)]
        [(< income bp2) (+ base1 (∗ (− income bp1) rate2))]
        [(< income bp3) (+ base2 (∗ (− income bp2) rate3))]
        [(< income bp4) (+ base3 (∗ (− income bp3) rate4))]
        [else (+ base4 (∗ (− income bp4) rate5))])))
Helper functions

There are many similar calculations in the tax program, leading to
the definition of the following helper function:

;;; (tax-calc base rate low high) calculates the total tax owed ...
;;; tax-calc: Num Num Num Num → Num
;;; requires base ≥ 0, rate ≥ 0, 0 ≤ low ≤ high
;;; Example:
(check-expect (tax-calc 1000 0.10 10000 10100) 1010)
(define (tax-calc base rate low high) (+ base (* rate (- high low)))

It can be used for defining constants and the main function.
(define base1 (tax-calc 0 rate1 0 bp1))
(define base2 (tax-calc base1 rate2 bp1 bp2))
(define base3 (tax-calc base2 rate3 bp2 bp3))
(define base4 (tax-calc base3 rate4 bp3 bp4))

(define (tax-payable income)
  (cond [(< income 0) 0] ; Not strictly necessary
        [(< income bp1) (tax-calc 0 rate1 0 income)]
        [(< income bp2) (tax-calc base1 rate2 bp1 income)]
        [(< income bp3) (tax-calc base2 rate3 bp2 income)]
        [(< income bp4) (tax-calc base3 rate4 bp3 income)]
        [else (tax-calc base4 rate5 bp4 income)]))
See HtDP, section 3.1, for a good example of helper functions.

Helper functions are used for three purposes:

1. *Reduce repeated code* by generalizing similar expressions.
2. *Factor out complex calculations*.
3. *Give names to expressions*.

Style guidelines:

- Improve clarity with short definitions using well-chosen names.
- Name all functions (including helpers) meaningfully; not “helper”.
- Purpose, contract, and one example are required.
Goals of this module

You should understand *Boolean data*, and be able to perform and combine comparisons to test complex conditions on numbers.

You should understand the syntax and use of a *conditional expression*.

You should understand how to write *check-expect* examples and tests, and use them in your assignment submissions.

You should be aware of other types of data (*symbols and strings*), which will be used in future lectures.
You should look for opportunities to use *helper functions* to structure your programs, and gradually learn when and where they are appropriate.
Boolean-valued Functions

1. The Boolean values are true and false. [2]
2. Some comparison operators are: =, <, ≤, >, ≥. [2-3]
3. Some Boolean connectives are: and, or, and not. [4]
4. and will produce true if all of its arguments are true. [5]
5. or will produce true if at least one of its arguments are true. [5]
6. Short-circuiting for and and or: Racket will only evaluate as many arguments as needed to determine the result. [6]
7. A predicate is a function that produces a Boolean result. [7]
Module 04 Summary

Strings and Symbols

8. Use symbol=? to test if one symbol equals another. [9]

9. A strings is a sequence of characters (i.e. text). [10]

10. There are many more operations on strings: joining, length, alphabetic order. [11]

11. Use symbols when you want to classify values into a few categories. [12]

12. There are many ways to check for equality: = for numbers, symbol=? for symbols, and equal? for any values. Use the most specific one possible. [13–14]
Module 04 Summary

Checking Conditions

13. Use \texttt{cond} to evaluate different expressions depending on the condition. [16]

14. Conditions are evaluated from the top to the bottom. [17]

15. When the first condition that evaluates to \texttt{true} is found, its corresponding expression is evaluated. [17]

16. If the condition \texttt{else} is reached, its expression will be evaluated. [17]

17. When a condition is reached, you know that all the conditions above it are false. You can use this fact to simplify the tests. [19]
Module 04 Summary

Testing

18. Test each cond question-answer pair. [22]

19. If there are boundary values, then the boundary value should be tested too. [22]

20. Tests that are not based on the details of the code (and are typically written first) are called **black-box tests**. [25]

21. Tests that are based on the details of the code, such as testing all the cond questions, are called **white-box tests**. [25]

22. **Helper functions** are used to reduce repeated code. [38]